Testing Patterns

Neelam Soundarajan[1], Jason O. Hallstrom[2], Adem Delibas[1], Guoqiang Shu[1]

[1] Computer Sc. & Engineering
Ohio State University
Columbus, OH 43210
{neelam, delibas, shug}@cse.ohio-state.edu

[2] Computer Science Dept.
Clemson University
Clemson, SC 29634
jasonoh@cs.clemson.edu

Abstract: *After over a decade of use, design patterns continue to find new areas of application. In previous work, we presented a* contract *formalism for specifying patterns precisely, and showed how the use of the formalism can amplify the benefits of patterns. In this paper, our goal is to enable practitioners to* test *whether their systems, as implemented, meet the requirements, as specified in the pattern contracts, corresponding to the correct usage of the patterns underlying the systems' designs. In our testing approach, corresponding to each design pattern, there is a set of what we call* pattern test case templates *(PTCTs). A PTCT codifies a reusable test case structure designed to identify defects associated with applications of the particular pattern. The test assertions in the PTCT are based on the requirements specified in the appropriate pattern contract. Next we present a process using which, given any system designed using the pattern, the system tester can generate a test suite from the PTCTs for that pattern that can be used to test the system for bugs in the implementation of the particular pattern. The process allows the system tester to tailor the test suite the needs of the individual system by specifying a set of* specialization *rules that are designed to reflect the structure and the scenarios in which the defects codified in the PTCTs are likely to manifest themselves in the particular system.*

1 Introduction

Design patterns [6, 3, 18] have had a profound impact on how software systems are built. This is not surprising since patterns capture the distilled wisdom of the software community, and provide proven solutions to recurring problems, solutions that can be tailored to the needs of individual systems. In previous work [20, 7, 21], we argued that in order to maximize the benefits of patterns, we must have precise specifications of patterns and how they are specialized in individual systems, and presented an approach to providing such specifications in the form of *pattern contracts* and *subcontracts*. A pattern contract captures the requirements and behavioral guarantees that apply to *all* uses of the particular pattern, while a subcontract characterizes how the pattern is specialized in a *particular application*. In this paper, we develop an approach to pattern-centric testing that designers, implementers, and testers of a given system can use to *test* their system against the contracts corresponding to the patterns used in the system's design, as specialized in the system subcontracts. Although some approaches to formalizing design patterns have been proposed [12, 5, 17, 4] (in addition to our own approach cited above), relatively little work seems to have focused on the question of *testing* a system to check whether it correctly implements the patterns underlying its design. We will discuss the few relevant elements of related work later in the paper. Since design-related bugs may not manifest themselves behaviorally until late in the development lifecycle, they tend to be among the most insidious and difficult to localize. The testing techniques and supporting software tools discussed in this paper will be of great value in reducing the likelihood of such bugs going undetected, and will aid in their localization.

Before considering some of the issues involved in pattern-centric testing, and our approach to addressing them, it may be useful to summarize our approach to formalizing patterns. Consider a pattern P. The contract for P consists of a set of *role-contracts*, one corresponding to each *role* of P, and a portion that corresponds to the pattern as a whole. The role-contract for a role R lists the *state* components of R, and specifies, in standard pre-/post-condition format, requirements that must be satisfied by the various methods of R. In a system designed using P, a class C *playing* the role R will typically provide *other* methods in addition to those "named" in R. If these "other" methods were to behave arbitrarily, then the intent of the pattern would be violated even if the methods corresponding to the named methods behaved according to their specifications in the role-contract. In order to eliminate this possibility, R's role-contract will also include an others specification that must be satisfied by all methods of C, except those corresponding to the ones named in the role-contract.

The portion of the pattern contract that corresponds to the pattern as a whole consists of an *invariant* over the (role) states of the various objects enrolled, at runtime, in an *instance* of the pattern. The invariant will be satisfied whenever control is not inside any of the methods of any of the participating objects. One important feature of the formalism is the notion of an *auxiliary concept*. An auxiliary concept is a relation over one or more states of one or more objects interacting with each other according to the pattern. Auxiliary concepts are *used* in the role-contracts and the pattern invariant, but their *definition* are *not* part of the pattern contract. Instead, definitions *tailored to particular systems*, are provided in the subcontracts corresponding to systems. The subcontract for a particular system also includes a set of *role-maps*, one corresponding to each class C playing a role R in the pattern, as applied in this system. The C-R-role-map specifies how the state, i.e., variables, of C map to the state of R (listed in its role-contract), which methods of C correspond to each of the named methods of R, etc. In effect, the role-map specifies how instances of C can be thought of *as* R-objects.

Consider the classic *Observer* pattern with its two roles, Subject and Observer. The complete contract for this pattern is presented elsewhere [7, 21]; here we consider some highlights. The intent of this pattern is to ensure that when the state of the subject[1] is *modified*, the observers "attached" to the subject are appropriately updated so that their states become *consistent* with the current state of the subject. But "modified" does not necessarily mean a change in any arbitrary bit or byte of the subject state; which modifications are important enough to require updates varies from system to system. Hence, we use an auxiliary concept, *Modified*(), a relation over two states of the subject, to distinguish "important" changes in the subject state from "unimportant" ones. Similarly, what it means for an observer state to be "consistent" with the subject state varies from system to system. Hence we use a second auxiliary concept, *Consistent*(), a relation over an observer state and a subject state, to represent this notion.

These two concepts allow us to precisely specify the various role methods of Subject and Observer, as well as the pattern invariant. For example, the specification of Observer.Update() has, as part of its post-condition, a clause that requires the state of the observer to be *Consistent*() with the current state of the subject it is observing. The other-method specification of this role is more interesting. The standard informal descriptions of the pattern suggest that unnamed methods not make *any* changes in the state of the observer. The post-condition, however, requires only that the state of the observer remain *Consistent*() with that of the subject.

As another example, the post-condition of Subject.Attach(), which is invoked to attach an object as a new observer, requires that a reference to the attaching observer be added to Subject._observers, which contains references to the subject's attached observers. It further requires that Attach() invoke Update() on the attaching observer to ensure that its state becomes *Consistent*() with the current state of the subject[2] . The post-condition of Attach() also includes a clause that requires the pre- and post-conditional states of the subject to satisfy the relation ¬*Modified()*. This clause ensures that the attached observers do not become *inconsistent* with the state of the subject, assuming they were *Consistent()* before the call to Attach(). Indeed, the invariant for the pattern simply states that the states of all the observers currently observing the subject are *Consistent*() with the subject state.

The contract for a pattern also specifies how a new instance of the pattern is created at runtime, and how objects may *enroll* in an existing instance to play a particular role. We refer the reader to [20, 7, 21] for full details. The key point to note for the purposes of this paper is that a pattern contract formalizes properties common across *all* applications of a given pattern; subcontracts capture the specializations appropriate to *particular* systems.

In this paper, we consider the following question: Given a contract for a pattern P, a system S designed using P, and the S-P-subcontract, how do we test the implementation of S to make sure that it meets all the requirements specified in P's contract, as specialized in the subcontract? Since the requirements specified in the pattern contract apply to *all* systems designed using the pattern, we adopt the following approach. We define a set of *pattern test case templates* (*PTCT*s) corresponding to each pattern P. Each PTCT defines an execution scenario involving one or more objects enrolled in various roles of P, a sequence of invocations on the role methods of these objects, and at key points, assertion statements derived from P's contract. The PTCT helps to test a system designed using P to see if the system meets the requirements specified in P's contract.

When designing the PTCTs corresponding to a particular pattern, we do not arrive mechanically at the results.

[1] We use names starting with uppercase letters, such as Subject, for roles. We use corresponding lowercase names, such as subject, to refer to the objects that play these roles.

[2] As we will see later, clauses of this kind are specified using the *trace* τ, a *ghost variable* [9] provided by the formalism that records the calls made during the execution of the method being specified.

Instead, we use our, and more importantly, the community's experience implementing P to identify *commonly* encountered errors. Consider again the *Observer* pattern. As we noted above, one of the requirements of the Subject.Attach() method is that the method invoke Update() on the attaching observer; otherwise, the state of the observer may not be *consistent* with the subject state and the pattern's intent could be violated. It seems that a common mistake in implementing the Observer pattern is to neglect to do this perhaps because it is not explicitly mentioned in the standard informal descriptions. The implementation of the pattern presented in [8], for example, does not contain such a call. Hence, it is important to design our PTCTs to check for this error in a variety of situations, with varying numbers of observers, with varying numbers of pattern instances, etc. Later in the paper we will consider another common error, again in the implementation of the Observer pattern, that may creep into systems, especially as they evolve over time. The PTCT we will construct will enable such faults to be detected and localized at an early stage.

Thus the essential idea behind our work is to enable testing of systems to identify mistakes in the implementation of the design underlying the system. And in performing such testing, our approach allows practitioners to take advantage of the experience and wisdom of the community as represented in the PTCTs. Thus the title of the paper may be interpreted in two different ways. On the one hand, the work is about testing systems against the contracts for the patterns underlying the systems' designs. On the other hand, the set of PTCTs for a given design pattern P can itself be considered as providing a *test pattern* for testing any system designed using P.

We should also note that the specific PTCTs that we present as examples are not cast in stone. Indeed, we expect that with additional information concerning common mistakes in implementing the associated patterns, these PTCTs will be refined and that additional PTCTs will be introduced. The main contribution of this paper, then, is the approach presented for testing systems to ensure the correct implementation of their underlying patterns — not the particular PTCTs presented as examples.

But a PTCT, by itself, is not an executable test case. It is a *generative template* that must be *specialized* to create test suites appropriate to particular systems. We consider various approaches to how this specialization might be achieved later in the paper. One approach that we will consider focuses on *automated* test case generation. In this approach, a test case generation tool accepts the contract for a pattern P as input, a PTCT for P, and a subcontract for a system constructed using P. Using the mappings specified in the subcontract, the tool generates a collection of test cases by replacing role method calls with calls to corresponding class methods, auxiliary concept relations (used in assertion statements) with appropriate definitions, etc. The advantage of such an approach is that it doesn't require any input from the system tester[3]. The disadvantage, however, is that such a tool makes it possible to overlook scenarios that might require especially thorough testing – to, for instance, account for the peculiarities of a particular system implementation. The only way to cater to such situations is to enable the system tester, based on his or her knowledge of the system internals, to decide the particular test cases to generate from the PTCT. To allow for this, we introduce the notion of a *test case specializer* (TCS). A system tester uses a TCS to specify how individual test cases should be generated from a given PTCT. In designing the TCSs for a given system, the tester will use his or her knowledge of the system, as well as considerations such as which parts of the system deserve more thorough testing. In a sense, the PTCTs corresponding to a pattern P can be thought of as the *testing contracts* that apply to all systems designed using P. The TCSs can be thought of as the *testing subcontracts* for a given system since they specify how the PTCTs should be specialized to obtain the actual test cases corresponding to that system.

An important question to consider in any testing work is the question of how to achieve suitable *coverage*, and how to measure coverage using suitable *metrics*. In standard software testing, metrics such as *code coverage*, *path coverage*, etc., are commonly used, and these have been refined in various ways for use in OO systems; see, for example, the detailed discussion in [2]. One approach that is of particular interest to us is *partition testing* [16, 23, 13]. As we will see, there is a natural correspondence between partition testing and our approach using PTCTs and TCSs. This correspondence suggests a notion of coverage for the type of testing we propose. Whether this is the most appropriate possible notion as well as possible alternative notions of coverage are points that we hope the discussion at the *SEW* will help clarify, assuming that the paper is accepted for the workshop.

A key consideration underlying our work is that it should be conveniently adoptable by software practitioners. To help with this, the notation we use is such that the PTCTs look similar to tests developed using standard unit testing packages such as *JUnit* [1]. The key distinction is that while these packages focus on units of *code*

[3] As we will see, this is not quite accurate; there are situations where some input may be required from the tester to help the tool generate the required objects, argument values etc. These points aside, however, the bulk of the work will be performed by the tool.

(such as a class method or an entire class), a set of PTCTs enables us to test a *unit of design*.

Paper Organization. Section 2 develops the essential notion of PTCT using examples based on the *Observer* pattern. Section 3 discusses how test cases may be generated from the PTCTs. Section 4 briefly discusses issues of coverage (and the associated metrics) in the context of pattern-centric testing. Section 5 briefly surveys elements of related work. Section 6 concludes with a summary of the approach, and pointers to future work.

2 Pattern Test Case Templates (PTCTs)

Many patterns are concerned with the *sequences of interactions* between various objects at appropriate points. PTCTs must be designed to test whether the actual interactions that occur match those dictated by the appropriate pattern contracts. Before we look at the construction of such PTCTs, we will consider elements of the *Observer* contract, including portions that specify such requirements. These requirements will be expressed using the *trace* τ, a *ghost variable* [9] provided by the formalism. In effect, when a method $m()$ starts execution, a corresponding trace τ, initialized to the empty sequence, is automatically created. Each call that $m()$ makes to a named method during its execution is recorded as an element of the sequence, and includes information about the object on which the method was was invoked, the name of the method, and information about the associated arguments and return values. A number of mathematical functions, some of which we will introduce later, simplify trace manipulation, access to individual trace elements, etc. Contract requirements concerning object interactions will be expressed in the form of appropriate conditions on the trace variables.

A portion of the *Observer* pattern contract appears in Fig. 1. Lines (3–4) list the auxiliary concepts used in the contract. In specifying the pattern invariant, we often need to refer to the objects enrolled in a given pattern instance. The contract formalism provides a special variable, players for this purpose. This variable maintains references to each of the enrolled "player" objects; they are stored in the order in which the objects enrolled. Thus, players[0] is the first object enrolled. In the case of instances of the *Observer* pattern, this object will be the subject; all other objects referenced by players correspond to the attached observers. Hence, the pattern invariant (lines 6–7) specifies that the state of the subject is always *Consistent*() with that of each observer.

Part of the Subject role contract appears next. The role state is specified (line 10) as consisting of the variable obs, used to store references to the observers currently attached to the subject. Next we have the specification of Attach(), one of the named methods of this role. The post-condition of Attach() imposes three requirements. First, (line 12) that the attaching observer not already be attached. Second, (line 13) that the method to add the attaching observer to obs where the "#" notation refers to the pre-conditional value of the variable, i.e., the value when the method started execution [22]. Finally, (line 14) that the method to not *modify*, in the sense of the concept *Modify()*, the state of the subject in question; and (line 14) that the method must have made one call[4] to a named method, this being to the Update() method on the attaching observer. The others specification (lines 16–19) requires that the other methods of this role not change the value of obs. It further requires that one of two conditions be satisfied. These methods must not *Modify()* the subject state, and not invoke any named methods; failing this, the Notify() method must be invoked. According to the specification of Notify() (omitted), this method will invoke Update() on each attached observer. Update() will, in turn, bring each observer into a state *Consistent()* with the new state of the subject. (Again, the specification is omitted.)

```
1  pattern contract Observer
2    concepts:
3      Consistent(Subject,Observer)
4      Modified(Subject,Subject)
5    invariant:
6      ∀ob∈players[1:]::
7          Consistent(players[0],ob)
8
9  role contract Subject
10   Set<Observer> obs;
11   void Attach(Observer ob):
12      pre: ob ∉ obs
13     post: (obs=(#obsU{ob}))
14        ∧ ¬Modified(#this,this)
15        ∧ (|τ|=1) ∧ (|τ.ob.Update|=1)
16   others:
17     post: (obs=#obs) ∧
18      ((¬Modified(#this,this) ∧ (|τ|=0))
19      ∨ ((|τ|=1) ∧ (|τ.this.Notify|=1)))
```

Fig. 1. Observer Pattern Contract (partial)

[4] $|\tau|$ denotes the length of τ, defined as the number of method calls made by this method during its execution. We often need the subsequence of τ consisting of elements corresponding to method calls on a particular object, say, o1; this is denoted $\tau.o1$. Similarly, $\tau.m1$ denotes the subsequence consisting of those elements of τ that correspond to calls to the method m1. Combining these, $\tau.o1.m1$ denotes the subsequence of calls to m1 on the object o1.

Given the requirements captured in the pattern contract, we must now develop appropriate PTCTs that can be used to test whether systems implemented using the Observer pattern satisfy these requirements. Or, more precisely, we must develop PTCTs that can be used to generate, using one of the approaches presented in the next section, specialized test cases that can be used for this purpose.

```
1  Subject s = new Subject();
2  Observer o = new Observer();
3  Set pre_obs = s.get_obs();
4  pre_obs.add(o);
5  Subject pre_s = s.clone();
6  tau.clear();
7  s.Attach(o);
8  Trace t1 = tau[0,t];
9  assert( (s.get_obs().equals(pre_obs)
10  && !Modified(pre_s, s)
11  && t1.length() == 1
12  && t1[0].m() == Update
13  && t1[0].ob() == o ) );
```

Fig. 2. PTCT (for Subject.Attach())

The main portion of a PTCT intended to test the behavior of `Attach()` appears in Fig. 2. We begin by creating a subject `s` (line 1), and an observer `o` (line 2). Next, we use a "*getter*" method provided by our testing framework, *JDUnit*[5], to retrieve the *pre-conditional* value of `s.obs` before the call to `Attach()` (line 7), and to store this value in `pre_obs`. (JDUnit provides appropriate *getter* methods for each role field, including *private* fields.) Next, we add the attaching observer to `pre_obs` (line 4) to simplify the expression of the assertion check corresponding to the post-condition of `Attach()` (lines 9–10). We additionally store the current state of `s` (line 5), since this is required to check that `Attach()` not *modify* `s` (according to an appropriate definition of *Modified*()[6].

[5]The name of the framework, currently under construction, is intended to draw a parallel to *JUnit*, the popular unit testing framework for Java; the additional *D* emphasizes that the framework is used to test units of *design*.

[6]A couple of points should be noted here. The *getter* methods are more involved than simply returning the current value of the field in question. In an actual system designed using the pattern, the class C playing a particular role R may provide a different set of fields from those specified in R's role contract. The subcontract for the system will specify the mapping from C's fields to those of R. The getter methods provided by *JDUnit* will make use of this mapping to *translate* the values of C's fields to the corresponding values of R's fields. Similarly, the `clone()` operation applied to `s` (line 5) will create an object of whatever class plays the `Subject` role in the particular application. Finally, JDUnit will use the concept definitions provided in the subcontract to evaluate assertions involving concept references, such as the assertion involving *Modified()* (line 9). We will return to these points in the next section.

Contract requirements involving *traces* require special treatment. As noted earlier, the contract formalism associates a trace variable τ, initialized to the empty sequence, with each method `m()`. Information about calls made by `m()` to `named` methods are recorded in τ; `m()`'s post-condition may impose suitable conditions on this variable. In those cases where a PTCT includes calls to methods with trace requirements, the PTCT will include `assert` statements that check relevant conditions on the associated trace variables. Indeed, a PTCT may include `assert` statements that require particular relations to hold across multiple trace variables, each associated with methods preceding the assertion check. The approach we use in *JDUnit* to handle such conditions is as follows. When the instantiated PTCT begins execution, the testing framework creates a trace object `tau`, initialized to the empty sequence. Each `named` method invoked during the execution of the PTCT is recorded as an element of `tau`, and includes information about the target object, the identity of the method invoked, etc. In addition, and this is the key difference from the use of τ in the formalism, each entry of `tau` itself includes the trace of methods executed during the associated call's execution. Hence, while there is only a single `tau` object, it maintains a *branching structure* corresponding to the *computation tree* rooted at the PTCT. Similar to the formalism, JDUnit provides simple functions for accessing and manipulating `tau` and its elements.

Immediately prior to the call to `Attach()`, we clear `tau` (line 6). This removes the trace entries associated with the preceding constructor calls (lines 1–2), as well as any calls to `named` methods introduced during the specialization process. (We will see an example of this type of specialization in the next section.) Hence, when control returns to the PTCT following the call to `Attach()`, `tau` contains a single entry, corresponding to this call. We store the trace in `t1` (line 8); `tau[0,t]` denotes the trace (`t`) associated with the zeroth element of `tau`. The `assert` that follows imposes appropriate conditions on this trace, based on the requirements specified in the pattern contract. The first two clauses require that `s._obs` be appropriately updated (line 9), and that `s` not be *Modified()* (line 10). The last three clauses address trace conditions imposed on `Attach()` (lines 11–13). Together, these clauses require that `Attach()` invoke exactly one `named` method, and that this call be to the `Update()` method of the attaching observer.

The `assert` statement is derived directly from the specification of `Attach()` included in the `Subject` role

contract shown in Fig. 1. The only additional derivation effort was the introduction of appropriate temporary objects to capture pre-conditional values referenced in the post-condition. The *pattern invariant* (lines 6–7, Fig. 1) could also have been checked as part of the assert statement. This would amount to checking that Attach() not only invokes Update() on the attaching observer, but also that the latter method appropriately updates the observer's state to be *Consistent()* with the subject. While the structure of the PTCT is dictated by relevant portions of the pattern contract, the test cases appropriate to a given system depend on the system's *implementation* details. Thus, the generated test cases will account for the specification *and* implementation details of the pattern. We will return to this point later in the paper. We should also note that when adding the invariant to the assert, we will have to replace references to elements of the players array by appropriate other variables. Thus players[0] will be replaced by the subject in question, etc.

```
1 Subject s = new Subject();
2 Observer o1 = new Observer();
3 Observer o2 = new Observer();
4 s.Attach(o1); s.Attach(o2);
5 Set pre_obs = s.get_obs();
6 Subject pre_s = s.clone();
7 tau.clear(); s.other();
8 Trace t1 = tau[0,t];
9 assert( s.get_obs().equals(pre_obs) &&
10  Consistent(s,o1) && Consistent(s,o2) );
```

Fig. 3. PTCT (for Subject.other())

Next consider the PTCT shown in Fig. 3, intended to test the behavior of Subject's other methods. We begin by creating a subject (s) and two observers (o1, o2) (lines 1–3), and attach both observers to the subject (line 4). We then save the pre-conditional value of s.obs (line 5) and the value of s as a whole (line 6). tau is then cleared, and *some* other method is invoked on s (line 7). Finally, the trace associated with the other call is saved (line 8), and the behavior of the method is checked against the requirements specified in the pattern contract (lines 9–10). The assert statement requires that s.obs be unchanged, and that both o1 and o2 be *Consistent()* with s.

In generating test cases for a particular system from this PTCT, the s.other() call will be replaced by calls to appropriate methods of the class playing the Subject role. Some of these methods may require arguments. As we will see in the next section, suitable values will be provided in such cases. At this point, however, the more important issue is the mismatch between the assert statement and the requirements specified in the *Observer* contract. According to the contract (Fig. 1), when s.other() terminates, either the state of s must not be *Modified()*, or the Notify() method must have been invoked. As we have already seen, this method will in turn invoke Update() on each attached observer, bringing the objects into states *consistent* with the new state of the subject. Hence, the assert included in the PTCT is a test of the expected *net* behavior of the participating objects. The advantage is that we are not using the trace when we do this; and if we adopt a similar approach uniformly in all our PTCTs (so, for example, we would have to rewrite the PTCT in Fig. 2 similarly), the implementation of JDUnit would be simplified since it would not have to create or, more importanly, update the trace. There is, however, a risk in adopting such an approach. If, for instance, an other method were to *modify* the state of the subject, neglect a call to Notify(), but by chance leave s in a state *Consistent()* with the states of o1 and o2, the design defect would go undetected.

In [19], we present a scenario in which this problem manifests itself during *system evolution*; we summarize the example here. A class S1 plays the Subject role, and provides two fields, f1 and f2. A change in either field is considered to be a *modification* of the subject according to the definition of *Modified()* supplied in the system's subcontract. A class O1 plays the Observer role. In an initial version of the system, O1 is interested only in the value of f1; changes in f2 are ignored. The Update() method of O1 uses an appropriate getter method to retrieve the value of f1, and then updates the observer's state to become *Consistent()* with the state of the subject. The *Consistent()* concept is defined suitably in the subcontract. The S1 class includes a bug that omits a call to Notify() when a change in f2 occurs. A PTCT similar to the one above will not detect this defect since each observer will remain *Consistent()* with the subject when the other method terminates — even if f2 has changed without a corresponding call to Notify().

During system evolution, O1 is modified so that the value of f2 becomes significant. Instances of O1 record the current value of f2 associated with their corresponding subject. Hence, the definition of *Consistent()* is suitably modified, as is the implementation of O1's Update() method. The new implementation retrieves the values of both f1 and f2, and updates the state of the observer appropriately. When the test case derived from the PTCT shown in Fig. 3 is executed, the assert statement will generate an error; the clause involving the *Consistent()* concept will be violated. The natural assumption is that the fault lies in an area affected by system evolution. In fact the defect lies in the original S1

class — in particular, the failure of its other() method to invoke Notify() when the value of f2 changes.

The solution is to revise the PTCT to fully test the interaction requirements specified in the pattern contract, rather than testing for *net* effects. More precisely, the last two clauses of the assert statement in Fig. 3 (line 10) should be replaced with the following conditions:

```
8 assert( ... clauses from Fig. 3 ...
9  && (!Modified(pre_s,s) ||
10   ((t1.length()==1)&&(t1.m()==notify))))
```

3 Generating Test Cases

Given PTCTs such as those in Fig. 2 and 3, how do we generate actual test cases corresponding to a given system built using the *Observer* pattern? Consider, for example, line 7 of Fig. 2. The method Attach() that is being invoked in this line may have an entirely different name in S. Indeed, even the classes playing the Subject and Observer roles are likely to have their own names appropriate for the application. In the *Hospital simulation* system presented in [21], for example, the Patient class plays the Subject role and the Nurse class plays the Observer role. The method that plays the Attach() role is Patient.addNurse(); the nurse being "attached" being passed as an argument to the method. As detailed in [21], all of this information is specified in the corresponding *subcontract*, in particular in the *role-map* that specifies how the Patient class plays the Subject role and how the Nurse class plays the Observer role. Thus it is straightforward for the *JDUnit* tool, given the PTCT and the subcontract, to construct a test case for the *Hospital* system by replacing the constructor calls in lines 1 and 2 by calls to constructors of the Patient and Nurse classes, the Attach() method call in line 7 by a call to Patient.addNurse() etc.

There are, however, some important questions in going from the PTCTs to the test cases. Again considering the *Hospital* system, not only does the Nurse class play the Observer role, so does the Doctor class. In other words, both nurse objects and doctor objects may be "attached" to a patient object. So in the PTCT in Fig. 2, should JDUnit replace the constructor call in line 2 by a call to the constructor of Nurse or Doctor? One possible solution, the one adopted in JDUnit, is to generate *two* test cases, one corresponding to each. For the PTCT in Fig. 3 which involves *two* observers, this would mean generating *four* different test cases corresponding to all possible combinations of o1 and o2 being doctor and nurse objects. But, in the *Hospital* system described in [21], one is not allowed to attach two doctor objects at the same time to a given patient object! In other words, an attempt to call addDoctor() on a patient object that already has a doctor will violate the pre-condition of this method and hence the method may behave in whatever way it chooses and this is *not* a bug in the system, despite the fact that, because of such behavior, the assert in the test case may fail. In order to address this type of problem, JDUnit inserts additional asserts in the test cases it generates immediately prior to calls to any methods. Each of these asserts checks that the pre-condition of the method being called is satisfied. If, when a test case is executed, such an assertion is *not* satisfied, that is recorded in the *log* maintained by the system; but the assertion failure does not indicate a failure of the test case[7].

Yet another question that we need to address in generating test cases from the PTCT concerns the call to the other() method that appears in the PTCT in Fig. 3 (line 7). In general, the class playing the Subject role may have *several* such methods. In that case, which of these methods do we call? One answer would, of course, be to generate multiple test cases, one corresponding to each of these methods. But what if some of these methods expect *additional* arguments? How do we generate suitable values for these additional arguments? Note that we cannot assign arbitrary values to these arguments since, as in the case considered in the last paragraph, the pre-condition of the method in question may expect these arguments to satisfy specific conditions. If the values we generate do not satisfy them, any assertion violation may not actually indicate a bug in the system at all. Hence, as in the earlier case, prior to such a call, JDUnit inserts the necessary check.

There is another important related issue. If the other() method in question does expect additional arguments, it would clearly be inadequate in general to generate only a handful of values for these arguments. For one thing, as we just noted, the generated values may not satisfy the method's pre-condition. Even if some of the values do satisfy the pre-condition, ensuring appropriate *coverage* of the range of possible behaviors that the method may exhibit will correspondingly require us to generate a sufficiently *wide range* of values –that satisfy the method's pre-condition– for these arguments. We will consider coverage questions in the next section.

Consider again the PTCT in Fig. 2. This, as we saw, is intended to test the behavior of the Attach() method.

[7] This is a distinction from the *JUnit* framework where every assertion violation is reported as a failure. In order to allow a different treatment for pre-condition violations, we use an additional method, preAssert, to specify such assertions.

In this PTCT, the method is invoked on a *freshly created* subject, passing a freshly created *observer* as argument. Suppose the tester of a given system such as the *Hospital* system suspects, based on his or her knowledge of the system internals, that this method (or rather the one(s) playing its role, as specified in the subcontract) functions correctly when the objects in question are in their newly instantiated states but may not be so well behaved if, before the call to Attach(), the objects had gone through certain state changes via certain calls to certain methods of their respective classes. In this case, the tester would clearly like to be able to generate one or more test cases that confirm or refute this suspicion. In order to enable this, we need to allow the tester to insert additional calls to various methods of the involved classes at various points in going from the PTCT to the test cases. If, however, the tester were to introduce arbitrary calls of this kind at inappropriate points, one or more of the asserts in the resulting test case might fail even though there is no bug in the system under test. For example, suppose the tester were to introduce a call to Detach() –or rather the method that plays this role– immediately following the call to Attach() in line 7 of this PTCT, passing the just attached observer as argument to this call to Detach(). Clearly the clause that appears in line 9 of the PTCT would be violated since s.obs will no longer contain a reference to the observer in question. To prevent such anomalies, we should only allow calls to other() methods to be introduced in this manner. These methods –assuming they satisfy the requirements in the others specification of the corresponding role– will not make such changes and hence the asserts in the test case must indeed be satisfied if the system correctly implements the pattern.

There are, however, situations where it is indeed necessary for the system tester to be able to insert, into the test case, calls to named methods beyond those that appear in the PTCT. For example, it may be that the tester, again because of knowledge of systems internals, feels that the Attach() method generally functions correctly. However, if before the call to Attach(), there are 10, or 20, or 30, ... observers already attached, then, perhaps because of the particular style of memory management used in the class, the method may fail to work correctly. Clearly in order to test such a case, the tester needs to be able to create a situation where, say, 10 observers are already attached, and then go through the code and the assertion checks as in the PTCT in Fig. 2.

In order to allow for all these considerations, we have currently adopted the following approach. A PTCT, in general, consists of an *initialization* segment followed by a *body* segment. For the PTCT in Fig. 2, the initialization segment consists of the first two lines, the rest being the body. The initialization segment will be followed by an assert. To simplify the presentation, we omitted this clause in the PTCTs in Figs. 2 and 3. For the first PTCT, the assert would require the observer o *not* be a member of s.get_obs(). For the second PTCT, the initialization segment consists of lines 1 through 4. The assert clause would require that o1 and o2 both be members of s.get_obs().

When generating test cases for a given system S, the system tester has the option of simply allowing the JDUnit tool to generate them from the PTCT using the information in the pattern contract, the subcontract for S, and the specifications of the methods of S. The tool will generate a relatively small number of test cases, replacing instances of various role objects with instances of the corresponding classes as specified in the subcontract, replacing calls to named methods with calls to the methods mapped, again as specified in the subcontract, to the named methods, replacing other() calls by calls to each of the other methods of the corresponding classes, etc.; and inserting the checks for pre-condition violations.

Alternately, and more commonly, the system tester will define a *test case specializer* (TCS). The TCS will first specify a (possibly empty) segment of code consisting of calls to named as well as other methods that the tester wants to have inserted into the test case following the initialization segment given in the PTCT. Thus, in the case of the PTCT in Fig. 2, the tester may, specify this part of the TCS to consist of a series of ten observer objects followed by calls to the Attach() method to attach each of these objects to the subject. The TCS notation (whose details we omit since they are still evolving) also allows the tester to specify, using a regular-expression notation, a number of alternative segments of code. In that case, the tool will generate a number of different test cases, one corresponding to each specified alternative. In each case, the JDUnit tool will insert the assert specified in the PTCT following the initialization segment. This will ensure that while the tester has the ability to initialize the test case in ways most suited for testing a given system, the basic intent of the PTCT is not violated.

The *body* of the PTCT has the structure we have discussed, but, may additionally define a number of points each of which is either an *any* extension point or an *other* extension point. The former has the syntax, L: ANY; where L is a unique label followed by an assert clause; the latter has the syntax L: OTHER;, L again being a

unique label and again followed by an `assert`. The intent is that, in the TCS, the tester may specify, for each such labeled point, a corresponding segment of code with the restrictiion that in the case of the *other* extension points, the code contain only calls to `other()` methods. The tool will insert this code in the generated test case (or test cases if more than such segment is specified), followed by the `assert` specified in the PTCT. In effect, these mechanism provides the tester with sufficient flexibility to generate, from the PTCT, the most suitable test cases for the particular system.

4 Coverage

Various *metrics* for measuring test coverage have been considered in the testing literature [2]. For approaches based on formal *specifications*, a metric based on *partitions* [16, 23, 13] is perhaps most suitable. The idea is as follows. Suppose the unit under test is just a simple function $f()$ that receives some arguments of simple types and returns a value of a simple type. The post-condition of $f()$ will typically be written as a number, say, n of *disjunctive* clauses, each corresponding to a particular set of possible values for the input arguments and specifying the condition that the output of $f()$ must satisfy for that case. These n sets of possible values of the input arguments give us the *specification partition* of the input domain. For convenience, we will name these sets $S_1, S_2, \ldots, S_n$. The *implementation* of $f()$, i.e., the code structure of $f()$ will provide us the *implementation* partition $I_1, \ldots, I_m$ of the input domain. In effect, each set in this partition corresponds to one *path* through the body of $f()$. Let us S_{jk} to denote the *intersection* of S_j and I_k. The key observation behind the approach [16] is that if there is a bug in the code, it must be along some particular path in the implementation of $f()$ and this should affect *all* the points in (at least) one of the S_{jk}. Thus if we ensure that our test cases include at least one point in each S_{jk}, we achieve complete coverage.

For our work, the pattern contract provides the specification partition. Thus, for example, the `others` specification in Fig. 1 contains two disjunctions corresponding to the two sets in its partition. The implementation partition would, of course, depend on the details of the system implementation. We are currently working on ways that a system tester can provide a specification of this partition. Given that, we can then check the test cases generated by JDUnit to determine how much coverage is achieved; and flag those S_{jk} sets that are not covered. The tester will then be able to revise the TCS to include the necessary additional test cases.

5 Related Work

Several authors [12, 5, 17, 4] have proposed approaches to formalizing patterns. None of these authors consider the issue of testing systems against specifications of patterns or their application in the system. McGregor *et al.* [10, 11] introduce the notion of a *test pattern*. This is somewhat similar to our PTCT in that it specifies a particular scenario of application objects interacting in some specific ways. However, there is no underlying specification of the *pattern* against which the behavior of the system is compared. Nor is there any notion of generating test cases for different systems from such a test pattern.

Reimer *et al.*'s [15] *SABER* system is designed to statically detect errors in large Java applications that use *particular* standard *frameworks*. Correct use of these frameworks require that certain methods be invoked in certain orders, that certain other methods *not* be at certain points, etc. Their system looks for violations of such requirements. One aspect of this work is that requirements involving, for example, a particular long sequence of method calls, are described in the form of *rules* in an XML database. Such an approach might also be applicable to the *static* portions of our PTCTs. Apart from these, there seems to be little work dealing with pattern-centric testing of systems.

6 Discussion

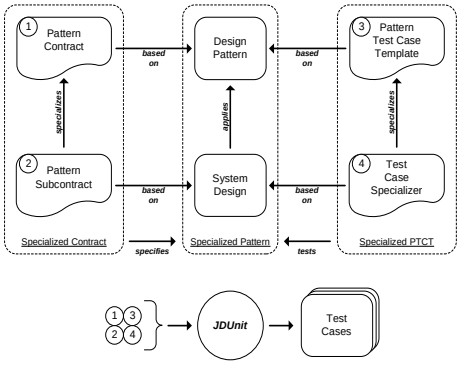

Fig. 4. Generating Test Cases

One of key guideline for using the JUnit framework is that the tests should be designed to test *behavior* rather than the methods of the class under test [14]. This also applies to our approach except that the behaviors in question are those corresponding to the use of a particular *pattern* in the system and hence, typically, involve multiple classes of the system, rather than the behaviors that

a single class is responsible for as is the case with standard unit testing. Fig. 4 pictorially depicts the relations between the different components of our approach.

Several issues remain to be addressed as we have seen in the previous sections. Perhaps the most important are those having to do with having a sufficiently flexible notation for expressing PTCTs as well as TCSs. We hope that the discussion at the workshop will also to address these issues satisfactorily.

References

[1] K. Beck, E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[2] R. Binder. *Testing object-oriented systems*. Addison-Wesley, 1999.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: A system of patterns*. Wiley, 1996.

[4] J. Dong, P. Alencar, and D. Cowan. A behavioral analysis approach to pattern-based composition. In *Proc. of the 7^{th} Int. Conf. on Object-Oriented Information Systems*, pp. 540–549. Springer, 2001.

[5] A.H. Eden. LePUS: a visual formalism for object-oriented architectures. In *Proc. of the 6^{th} World Conference on Integrated Design and Process Technology*, pp. 149–159. Computer Society, 2002.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.

[7] J.O. Hallstrom, N. Soundarajan, and B. Tyler. Amplifying the benefits of design patterns. In J. Aagedal and L. Baresi, editors, *The 9^{th} Int. Conf. on Fundamental Approaches to Softw. Eng. (FASE)*, pp. 214–229. Springer, 2006.

[8] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. of OOPSLA*, pp. 161–173. ACM, 2002.

[9] C. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1990.

[10] J McGregor. Testpatterns: Please stand by. *Journal of Object-Oriented Programming*, 12:14–19, 1999.

[11] J McGregor, D Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison Wesley, 2001.

[12] T. Mikkonen. Formalizing design patterns. In *Proceedings of 20th ICSE*, pp. 115–124. IEEE Computer Society Press, 1998.

[13] S Ntafos. Comparisons of random, partition, and proportional partition testing. *IEEE Trans. on Softw. Eng.*, 27(10):949–960, 2001.

[14] J Rainsberger. *JUnit Recipes*. Manning, 2005.

[15] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. Johnson, A. Kershenbaum, and L. Koved. "SABER": smart analysis based error reduction. In *Proc. of "ISSTA '04"*, pp. 243–251. ACM Press, 2004.

[16] D Richardson L Clarke. Partition analysis. *IEEE Trans. on Softw. Eng.*, 11(12):1477–1490, 1985.

[17] D. Riehle. Composite design patterns. In *Proc. of OOPSLA*, pp. 218–228. ACM, 1997.

[18] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. *Pattern-oriented software architecture: Patterns for concurrent and networked objects*. Wiley, 1996.

[19] N. Soundarajan and J. Hallstrom. Pattern-based system evolution: A case-study. In K. Zhang, G. Spanoudakis, and G. Visaggio, editors, *18th Int. Conf. on Softw. Eng. and Knowledge Eng. (SEKE)*, pp. 321–326. Knowledge Systems Institute, 2006.

[20] N. Soundarajan, J.O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In A. Finkelstein, J. Estublier, D. Rosenblum, editors, *Proc. of 26th Int. Conf. on Softw. Engineering (ICSE)*, pp. 666–675. Computer Society, 2004.

[21] B Tyler, J Hallstrom, and N Soundarajan. A comparative study of monitoring tools for pattern-centric behavior. In M Hinchey, editor, *Proc. of 30th IEEE/NASA Sofware Engineering Workshop (SEW-30)*. IEEE-Computer Society, 2006.

[22] J. Warmer and A. Kleppe. *The Object Constraint Langauge*. Addison-Wesley, 1999.

[23] E Weyuker and B Jeng. Analysing partition testing strategies. *IEEE Trans. on Softw. Eng.*, 17(7):703–711, 1991.